Science of Computer Programming 10 (1988) 65-85
North-Holland

NONDETERMINISTIC DATA FLOW PROGRAMS: HOW TO AVOID THE MERGE ANOMALY

Manfred BROY

Fakultät für Mathematik und Informatik, Universität Passau, 8390 Passau, Fed. Rep. Germany

Communicated by M. Sintzoff
Received October 1985

Abstract. A simple programming language for the description of networks of loosely coupled, communicating, nondeterministic agents is introduced. Two possible graphical interpretations are discussed: finite cyclic and infinite acyclic, tree-like graphs. Operational semantics for such graphs is defined by computation sequences. The merge anomaly is described, analysed and explained. Two fixed-point semantics are defined in a denotational style, one that avoids the merge anomaly, and another one that includes the merge anomaly, and they are proved to be consistent with the resp. operational definitions. Both definitions are compared and analysed.

1. Introduction

Families of stream-processing functions provide a nice, simple, and elegant model for systems of loosely coupled (asynchronous), communicating processes. However, for modelling a number of classical problems appearing in concurrent programming, one is interested in considering also nondeterministic stream-processing functions such as for instance the function "merge" nondeterministically merging two streams. But the inclusion of nondeterministic stream processing functions leads to some intricate problems, when trying to give a denotational semantics to such systems. One of them is the so-called merge anomaly.

Since mentioned the first time in [8], the merge anomaly deserved much attention in the literature. Several papers have been published describing and investigating the merge anomaly, and proposing solutions to it, sometimes leading to quite complicated constructions.

In the following sections we introduce a very simple language for defining nondeterministic networks of computing agents. Two graphical interpretations for the language are considered: finite cyclic graphs and infinite acyclic, tree-like graphs that arise from the finite ones by unfolding all cycles. Both interpretations are feasible and lead to the same extensional ('input/ouput') behaviour for the networks in the case of deterministic agents, if a straightforward operational semantics is defined for such (data flow) graphs. However, for nondeterministic agents we get

distinct operational behaviours for the two graphical interpretations and the corresponding operational semantics.

For both graphical interpretations that we consider we give straightforward denotational semantics for our language using techniques from fixed-point theory as suggested in [6]. It is shown that the merge anomaly comes from the discrepancy that arises, when people take the more suggestive graphical interpretation using finite cyclic graphs and the corresponding operational semantics, but a denotational semantics that corresponds to infinite acyclic, tree-like graphs. Finally we compare our proposal for avoiding the merge anomaly to others advocated in the literature.

2. Streams

In a net of communicating agents the communication flowing from agent a to agent b can be represented by a finite or infinite sequence of atomic data. So for giving a semantics to communicating agents we introduce the notion of streams. Basically a stream is defined by a sequence of atoms. Now let $ATOM$ be some given set of atomic values (including the natural numbers and the truth values for instance). By $ATOM^{\perp}$ the classical flat domain over $ATOM$ is denoted, i.e. the partially ordered set with just $\perp$ as the least element and all other elements incomparable. By $ATOM^*$ we denote the finite sequences from $ATOM$, by $ATOM^{\infty}$ the infinite sequences. Then the set of streams is defined by

$$STREAM(ATOM) \stackrel{\text{def}}{=} ATOM^* \cup ATOM^{\infty}.$$

With this definition $STREAM(ATOM)$ forms an algebraic domain (i.e. a complete partially ordered set where all elements can be represented by their finite approximations) consisting of the union of all finite partial streams with all infinite (and total) streams, if we use the ordering for streams s_1, s_2, defined by

$$s_1 \sqsubseteq s_2 \Leftrightarrow s_1 \text{ is prefix of } s_2.$$

By $\langle a \rangle$ the one-element sequence is denoted consisting just of the atom a from $ATOM$. The symbol '++' is used to denote the usual concatenation of sequences to sequences. By '&' we denote the operator adding an atomic element as first element to a stram, i.e.

$$a \,\&\, s = \langle a \rangle ++ s \quad \text{for } a \in ATOM, \qquad \perp \,\&\, s = \varepsilon$$

is assumed. By ε the empty stream is denoted. Note that '&' is leftstrict, i.e. $a \,\&\, \varepsilon \neq \varepsilon$ although ε is the least element. For streams we use the following two continuous functions:

$$first : STREAM(ATOM) \to ATOM^{\perp},$$

$$rest : STREAM(ATOM) \to STREAM(ATOM)$$

which are defined by (for $a \in ATOM$)

$$first(\varepsilon) = \perp, \qquad first(a \,\&\, s) = a,$$

$$rest(\varepsilon) = \varepsilon, \qquad rest(a \,\&\, s) = s.$$

Streams are a very basic notion in concurrent communicating systems. In systems based on shared memory one has to consider streams of states, in tightly coupled systems one has to consider streams of actions.

3. The language

Throughout this paper we consider a language of *data flow programs* with the following simple syntax:

$$\langle net\rangle ::= [\{\langle equation\rangle\,,\}^*(\langle tuple\rangle)],$$

$$\langle equation\rangle ::= \langle tuple\rangle = \langle function\rangle(\langle tuple\rangle),$$

$$\langle tuple\rangle ::= \langle id\rangle\{\,,\langle id\rangle\}^*,$$

$$\langle function\rangle ::= merge \mid \langle fid\rangle$$

where $\langle id\rangle$ stands for some given set of identifiers for streams, and $\langle fid\rangle$ denotes a set F of function identifiers. For every f from F we assume that there are $n, m \in \mathbb{N}$ and a continuous function

$$f' : STREAM(ATOM)^n \to STREAM(ATOM)^m.$$

The pair (n, m) is called the arity of f and we write

$$arity(f) = (n, m).$$

In particular, $arity(merge) = (2, 1)$. A well-formed data flow program is a program P of the form

$$[x_1 = f_1(y_1), \ldots, x_k = f_k(y_k), y_{k+1}],$$

where we assume that for i, $1 \leq i \leq k$, the function symbol f_i has arity (n_i, m_i), that x_i is an m_i-tuple of identifiers from $\langle id\rangle$, and that y_i is an n_i-tuple of identifiers from $\langle id\rangle$. Of course we assume that all components of the x_i are distinct identifiers.

By $ID(P)$ we denote the set of identifiers from $\langle id\rangle$ occurring in P. If there exists an identifier z for which $z \neq x_i$ for $1 \leq i \leq k$ and $y_i = z$ for some i, then z is called *input-port*. By $IN(P)$ we denote the set of input-ports of P. The identifiers occurring in y_{k+1} are called output-ports. The set of *output ports* of P will be denoted by $OUT(P)$.

4. Networks as graphs

With every program P, i.e. with every syntactic object of the syntactic unit $\langle net \rangle$, we may associate a (data flow) graph. Basically with the recursive equations in P we may associate either an infinite acyclic, tree-like graph with P or a finite cyclic one. We are now going to define this relationship formally and illustrate this by an example which will also be used for explaining the merge anomaly.

For giving the example we assume a function symbol $g \in F$ for the continuous function g' with $arity(g) = (1, 1)$ specified by (let $a \neq b \neq c \neq a$ hold)

$$g'(\varepsilon) = \varepsilon,$$

$$g'(a \,\&\, s) = c \,\&\, g'(s),$$

$$g'(x \,\&\, s) = b \,\&\, g'(s) \quad \text{for } x \neq a \text{ and } x \neq \bot.$$

Now we consider the program MA:

$$[y = merge(x, z),\ z = g(y),\ (z)].$$

Here x is the only input stream and z the only output stream.

4.1. *Data flow graphs*

Let $VERTEX$ be a given set of vertices, and ARC a given set of arcs. An *interpreted (data flow) graph* is a tuple $(V, A, I, O, in, out, label)$ where

$$V \subseteq VERTEX, \quad A \subseteq ARC, \quad I \subseteq A, \quad O \subseteq A,$$

$$in : V \to A^*,$$

$$out : A \setminus I \to (V \times \mathbb{N}),$$

$$label : V \to (\langle fid \rangle \cup \{merge\}),$$

such that for all $v \in V$,

$$arity(label(v)) = (n, m) \Rightarrow in(v) \in A^n,$$

$$arity(label(v)) = (n, m) \wedge out(a) = (v, i) \Rightarrow 1 \leqslant i \leqslant m$$

$$arity(label(v)) = (n, m) \Rightarrow \forall i, 1 \leqslant i \leqslant m : \exists a \in A : out(a) = (v, i).$$

Intuitively a data flow graph can be considered as a network of processors that are associated with the vertices each of which is realizing a particular stream-processing function (indicated by the respective label). These processors are connected by directed lines (the arcs) over which finite or infinite sequences of data are flowing. In the formal definition above the set V denotes the vertices of the data

flow graph, the set A denotes the arcs of the data flow graphs, I denotes the set of input arcs, O denotes the set of output arcs. The mapping *in* indicates the tuple of input arcs for the respective vertex; the mapping *out* indicates which vertex is the source of the respective arc and the position of this output line for that vertex. The mapping *label* indicates which stream-processing functions are associated with the vertices. An important question is that of the transformation of data flow networks and of structural equivalence of two data flow networks: under which circumstances can we replace a network by a smaller one without affecting its correctness. For this reason we introduce the notion of a graph morphism.

A *graph morphism* between two interpreted data flow graphs $G1=(V1, A1, I1, O1, in1, out1, label1)$ and $G2=(V2, A2, I2, O2, in2, out2, label2)$ consists of a pair of mappings between the vertices and arcs of the graphs:

$$\alpha_V: V1 \to V2, \qquad \alpha_A: A1 \to A2$$

such that

$$\alpha_A^*(in1(v)) = in2(\alpha_V(v)),$$

$$label1(v) = label2(\alpha_V(v)),$$

$$out1(a) = (v, i) \Rightarrow out2(\alpha_A(a)) = (\alpha_V(v), i),$$

$$\forall x \in I1:\ \alpha_A(x) = x \ \wedge\ x \in I2,$$

$$\forall x \in O1:\ \alpha_A(x) = x \ \wedge\ x \in O2.$$

Here α^* simply denotes the elementwise extension of α to tuples. A graph morphism is called *surjective*, if both α_A and α_V are surjective.

A graph morphism may not be injective. Then certain vertices or arcs are identified and we just may use one processor in $G1$ instead of a couple of processors in $G2$.

A *state* of a data flow graph is a total function

$$\sigma: A \to STREAM(ATOM).$$

A state σ is call *initial* if $\sigma(a) = \varepsilon$ for all $a \in A \setminus I$.

A finite sequence $(a_0, \ldots, a_j)$ of arcs is called a *path* in a graph, if for all i, $0 \leqslant i \leqslant j$, there exists a vertex v such that $in(v)$ contains a_i and $out(a_{i+1}) = (v, l_i)$ for some l_i. A nontrivial (i.e. $0 < j$) path is called a *cycle* if $a_0 = a_j$. A graph is called *acyclic*, if all paths in g are not cycles. A data flow graph is called *sharing-free*, if for every *arc* $a \in A \setminus I$ there exists at most one vertex $v \in V$ where a occurs within $in(v)$ and a occurs only at one position in $in(v)$ and the vertices in O do not occur in $in(v)$ for any v.

For the given program P:

$$[x_1 = f_1(y_1), \ldots, x_k = f_k(y_k), y_{k+1}],$$

we call a graph $g=(V, A, I, O, in, out, label)$ an *associated graph* iff there exist

bijective mappings

$$vertex: \{1, \ldots, k\} \to V, \qquad arc: ID(P) \to A$$

such that

$$\{arc(x): x \in IN(P)\} = I, \qquad \{arc(x): x \in OUT(P)\} = O,$$

$$\forall i, 1 \leq i \leq k: \ label(vertex(i)) = f_i,$$

$$\forall i, 1 \leq i \leq k: \ arc^*(y_i) = in(vertex(i)),$$

$$\forall i, j, 1 \leq i \leq k, 1 \leq j \leq m_i: \ out(arc((x_i)_j)) = (vertex(i), j).$$

Note that the associated graph is uniquely determined up to graph isomorphisms.

4.2. Operational semantics of data flow graphs

Given an interpreted (data flow) graph G and an initial state σ_0, with $\sigma_0(a) = \varepsilon$ for all $a \notin I$, a *computation sequence* in g starting in σ_0 is a sequence $\{\sigma_i\}_{i \in \mathbb{N}}$ of states where for all $a \in A$,

$$a \in I \Rightarrow \sigma_{i+1}(a) = \sigma_0(a),$$

$$a \notin I \Rightarrow \sigma_{i+1}(a) = s_j,$$

where s_j is defined as follows. Let

$$output(a) = (v, j), \qquad label(v) = f,$$

$$input(f) = (a'_1, \ldots, a'_n), \qquad arity(f) = (n, m).$$

If $f \in \langle fid \rangle$, we define:

$$(s_1, \ldots, s_m) = f'(\sigma_i(a'_1), \ldots, \sigma_i(a'_n)),$$

and if $label(v) = merge$ (note that then $j = 1$ holds), then there exists $d \in \{1, 2\}^\infty$ such that for all i,

$$\sigma_{i+1}(a) = sched(\sigma_i(a'_1), \sigma_i(a'_2), d).$$

Here the continuous function

$$sched: STREAM(ATOM) \times STREAM(ATOM) \times \{1, 2\}^\infty \to STREAM(ATOM)$$

is specified by

$$sched(s1, s2, 1 \,\&\, d) = first(s1) \,\&\, sched(rest(s1), s2, d),$$

$$sched(s1, s2, 2 \,\&\, d) = first(s2) \,\&\, sched(s1, rest(s2), d).$$

Note that for simplicity we have chosen a definition of *merge* such that *merge* is neither fair nor nonstrict. According to this definition we obtain the following lemma:

Lemma 4.1. *A computation sequence for an interpreted graph is a chain.* □

Thus the state σ, where

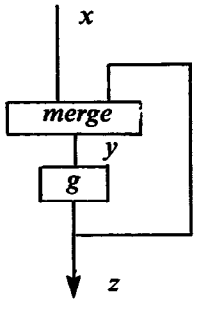
$$\sigma = \text{lub}\{\sigma_i\},$$

is well-defined. It is called the *result* of the computation sequence.

Note that this is a rather abstract version of an operational semantics. But it is not very difficult to give a much more concrete one based on firing rules for the vertices that is equivalent to our abstract one.

4.3. Net programs as finite cyclic data flow graphs

A data flow program may be interpreted as a finite cyclic graph, if we take the associated graph as graphical representation. So for the example program *MA* we obtain



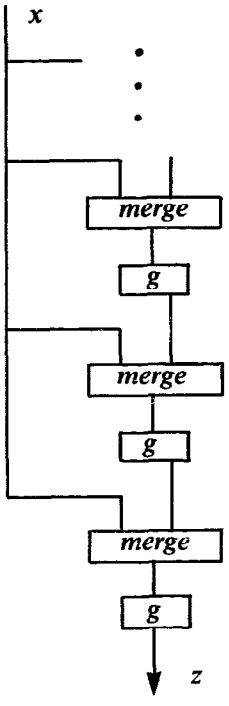
The associated graph is determined up to isomorphism and it is suggestive to use it as the graphical representation of a data flow program.

4.4. Net programs as infinite acyclic sharing-free data flow graphs

A possibly infinite graph can be associated with a program according to the techniques of [11] by eliminating sharing by copying unfolding all cycles infinitely often. For convenience we associate such an infinite graph to a data flow program by the following definition.

A graph g_∞ is called (free, infinite) *acyclic* graph associated with a given program *P* iff g_∞ is acyclic and sharing-free and there exists a surjective graph morphism from g_∞ onto the associated graph.

For our example program *MA* we obtain an infinite acyclic sharing-free graph



Note that here an infinite number of copies of the stream associated with x are needed.

5. The merge anomaly

The merge anomaly has been mentioned the first time in [8]. It shows a basic contradiction between the 'natural' operational understanding of data flow programs as cyclic graphs with nondeterministic stream-processing functions associated with their nodes and a denotational semantics associating set-valued functions with the nodes and sets of streams with the arcs of such data flow programs.

If we consider the finite data flow graph associated with the program *MA* in the previous section, we initially have the situation (for the initial state with input $x = a\&\varepsilon$)

$$(x, y, z) := (a\&\varepsilon, \varepsilon, \varepsilon);$$

then *merge* may fire to the left (to the right it is just the identity) and we get

$$(x, y, z) := (a\&\varepsilon, a\&\varepsilon, \varepsilon),$$

now g may fire leading to

$$(x, y, z) := (a\&\varepsilon, a\&\varepsilon, c\&\varepsilon),$$

now the network outputs c and *merge* may fire to the left (now to the right gives just the identity) leading to

$$(x, y, z) := (a\&\varepsilon, a\&c\&\varepsilon, c\&\varepsilon),$$

then g may fire again and we obtain

$$(x, y, z) := (a\&\varepsilon, a\&c\&\varepsilon, c\&b\&\varepsilon).$$

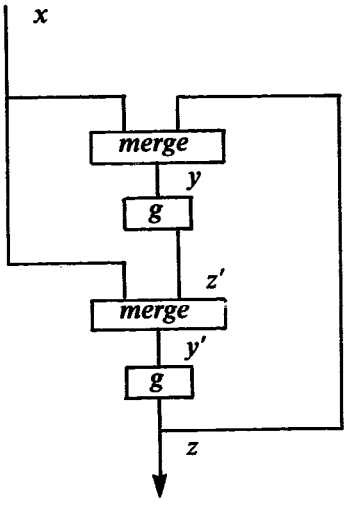
Now the network outputs b again and *merge* may fire again....

So the only first output the network may produce (if any) is c. Of course (as long as *merge* is not nonstrict), the network may also give ε as output, i.e. no 'defined' output at all.

For giving a denotational semantics to nondeterministic data flow programs, one could associate set-valued functions with nodes and sets of streams with the arcs. Now assume X, Y, Z are the sets of streams associated with the identifiers x, y, z resp. of the data flow program *MA*. In particular, the set Z is the set of possible outputs of the program. Assume there is some stream $s_z \in Z$ with $first(s_z) = c$. Then obviously there should be some stream s_y in Y with $first(s_y) = c$, too, since Y is the set resulting by merging the elements from Z with the elements from X. But then there has to be some $s'_z \in Z$ with $s'_z = g(s_y)$ and thus $first(s'_z) = b$. So b may be the first output of the program. This is obviously a contradiction to the operational semantics above. This surprising contradiction is the puzzle that is called the *merge anomaly*. But what is the reason for this contradiction?

A first hint where this fictitious contradiction comes from can be obtained by analysing the operational behavior of the infinite graph. Here b may well be an output of the graph if, for instance, the second last merge node takes its first input from the left, the second last g produces some c, then the last merge has only to take its input from the right, g produces b from the input c and b is the first output.

Obviously for nondeterministic data flow programs it makes an essential difference, whether we associate an infinite, acyclic, sharing-free graph or a finite, cyclic graph with them. Both possibilities are feasible, but lead to different input/output behaviors. So they need distinct fixed-point techniques as is demonstrated in the preceding section. Note that even the data flow graph



that arises by 'unfolding' z and then y in MA, introducing new auxiliary identifiers z' and y', and then exchanging z and z' on the left-hand side of the equations (such a transformation would be correct if *merge* would be a deterministic function) leading to the program

$$[y = merge(x, z), z = g(y'), y' = merge(x, z'), z' = g(y), (z)],$$

may (according to our operational semantics) produce b as output for the input $x = a\&\varepsilon$. This shows that in spite of the equality sign we must *not* simply replace the left-hand side of an equation for streams by the respective right-hand side of a recursive stream equation. This shows that we cannot use a straightforward fixed-point definition for such equations.

6. Denotational semantics of data flow programs

In this section we give two denotational definitions of semantics for data flow programs. One based on set-valued functions and on power domains and the other one based on the idea of sets of continuous functions.

To start with we introduce a simple generalization of power domains (cf. [13, 15]).

6.1. Basic definitions of power domains

Let DOM be a countably algebraic domain, i.e. DOM is a complete partially ordered set of all elements of which can be represented as least upper bounds of their set of finite approximation. Moreover we assume, that the set of finite elements is countable. For $S, S1, S2 \in DOM$ the following preordering (called 'Egli-Milner' ordering) is used (cf. [13, 15]):

$$S1 \sqsubseteq_{EM} S2 \text{ iff } \forall x \in S1 \exists y \in S2: x \sqsubseteq y \wedge \forall y \in S2 \exists x \in S1: x \sqsubseteq y.$$

Over nonflat (nondiscrete) domains like $STREAM(ATOM)$ this relation just defines a preordering. Let $FDOM$ denote the set of finite elements from DOM.

As the power domain $P_{EM}(DOM)$ over DOM we consider just a subset of the powerset $P(DOM)$. At first we introduce a equivalence relation on $P(DOM)$:

$$S1 \approx S2 \text{ iff for all } S \subseteq FDOM, S \text{ finite}: S \sqsubseteq_{EM} S1 \Leftrightarrow S \sqsubseteq_{EM} S2.$$

This way we obtain classes of $\sim$–equivalent sets. Now let $P_{EM}(DOM)$ be defined as the set of sets from $P(DOM)$ that are maximal in the inclusion ordering in their $\sim$–equivalence classes. $P_{EM}(DOM)$ will be used as the power domain in the following. By

$$C_{EM}: P(DOM) \Rightarrow P_{EM}(DOM)$$

we denote the mapping that associates with every set $S \subseteq DOM$ its power domain representation. More explicitly one has

$$FS: (\langle fid \rangle \cup \{merge\}) \to P_{EM}(STREAM(ATOM)^n) \to P_{EM}(STREAM(ATOM)^m),$$

defined by

$$f \in \langle fid \rangle \Rightarrow FS[f](S) = \{f'(s)\colon s \in S\},$$

$$FS[merge](S) = \{sched(s1, s2, d)\colon d \in \{1,2\}^{\infty} \wedge (s1, s2) \in S\}$$

where *sched* is defined as above.

With every program P:

$$[x_1 = f_1(y_1), \ldots, x_k = f_k(y_k), y_{k+1}],$$

we associate now a functional $TS[P]$:

$$TS : \langle net \rangle \to PENV(ID(P)) \to PENV(ID(P)),$$

where

$$TS[P](E)(x) = C_{EM}(\{E[S_1/x_1, \ldots, S_k/x_k](x)\colon S_i \in FS[f_i](E(y_i))\}).$$

Now we can define $BS[P]$ as the $\sqsubseteq_{EM}$-fixed point of the equation

$$BS[P](E)(x) = TS[P](BS[P])(E)(x).$$

If we want to hide the internal streams (treating the corresponding stream-identifiers as 'locally defined' or 'bound'), the semantics of P is determined by the mapping

$$BSH : \langle net \rangle \to PENV(ID(P)) \to PENV(ID(P)),$$

defined by

$$BSH[P](E) = E[BS[P](E)(y_{k+1})/y_{k+1}].$$

Treating data flow programs as set-valued functions corresponds to the graphical interpretation considering infinite acyclic, sharing-free graphs: A 'nondeterministic' stream has many distinct (i.e. a set of) instantiations in one computation. The correctness of this semantic definition (modulo the mentioned identifications) with respect to the operational one for the associated acyclic, sharing-free data flow graph can be seen by the following theorem:

Theorem 6.1. *Let P be a program and g an associated acyclic, sharing-free graph, i.e. there exists a surjective mapping*

$$\alpha : A \to ID(P),$$

where A denotes the arcs of g. Let furthermore S be the set of results of computation sequences in g starting from the initial state σ_0*; then*

$$BS[P](E_{\sigma_0})(x) \sqsubseteq_{EM} \{s \in E_{\sigma}(x)\colon \sigma \in S\} \sqsubseteq_{EM} BS[P](E_{\sigma_0})(x)$$

where for arbitrary states σ *the nondeterministic environment* E_{σ} *is defined by*

$$E_{\sigma}(x) = \begin{cases} \{\varepsilon\} & \text{if } x \notin ID(P), \\ \{\sigma(a)\colon x = \alpha(a)\} & \text{if } x \in ID(P). \end{cases}$$

Sketch of proof. Let S^* be the set of computation sequences in g. We have

$$TS[P]^i(E_{\sigma_0})(x) \sqsubseteq_{EM} \{s \in E_{\sigma_i}(x): \{\sigma_i\}_{i\in\mathbb{N}} \in S^*\} \sqsubseteq_{EM} TS[P]^i(E_{\sigma_0})(x),$$

since for each finite, cyclic path c in the associated graph we can find a noncyclic path of the same length which is mapped onto c and vice versa. So for every i we may distribute the set of streams associated with an identifier arbitrarily over the infinite graph. Since $TS[P]$ is continuous we have

$$\begin{aligned}
&BS[P](E_{\sigma_0})(x)\\
&\quad = \quad \sqsubseteq_{EM}\text{-lub}\{TS[P]^i(E_{\sigma_0})(x)\}\\
&\sqsubseteq_{EM} \sqsubseteq_{EM}\text{-lub}\{C_{EM}(\{s \in E_{\sigma_i}(x): \{\sigma_i\}_{i\in\mathbb{N}} \in S^*\})\}\\
&\sqsubseteq_{EM} \{s \in E_\sigma(x): \sigma \in S\}\\
&\sqsubseteq_{EM} \sqsubseteq_{EM}\text{-lub}\{C_{EM}(\{s \in E_{\sigma_i}(x): \{\sigma_i\}_{i\in\mathbb{N}} \in S^*\})\}\\
&\sqsubseteq_{EM} \sqsubseteq_{EM}\text{-lub}\{TS[P]^i(E_{\sigma_0})(x)\}\\
&\quad = \quad BS[P](E_{\sigma_0})(x). \qquad \square
\end{aligned}$$

According to the construction of the powerdomain we immediately obtain

Corollary 6.2. *Under the assumptions of the theorem we have*

$$BS[P](E_{\sigma_0})(x) = C_{EM}(\{s \in E_\sigma(x): \sigma \in S\}).$$

If $\{s \in E_\sigma(x): \sigma \in S\}$ *is closed (w.r.t. lubs) and convex, then we even have*

$$BS[P](E_{\sigma_0})(x) = \{s \in E_\sigma(x): \sigma \in S\}. \qquad \square$$

Basically this shows that under this interpretation of data flow programs, the nondeterministic choice for some stream associated with some identifier x can be done 'individually' for every occurrence of x. This is of course in contradiction with the inuitive notion of communicating systems where multiple occurrences of identifiers for streams correspond to the sharing of streams which should have one determined identity in every instance of a computation of the data flow program.

6.3. Nets as sets of functions

For giving a denotational semantics to data flow programs we introduce the well-known concept of environments. For every set of identifiers ID we introduce

$$ENV(ID) = \{e: ID \to STREAM(ATOM)\}.$$

As usual we denote the componentwise change of an environment e by $e[s/x]$ where x is some identifier and s is some stream. Formally we have

$$e[s/x](y) = \begin{cases} s & \text{if } x = y,\\ e(y) & \text{otherwise.}\end{cases}$$

If s is an *n-tuple* $(s_1, \ldots, s_n)$ of streams and x a tuple of n distinct identifiers $x_1, \ldots, x_n$ we write

$$e[s/x] \quad \text{for} \quad e[s_1/x_1] \ldots [s_n/x_n]$$

and also

$$e[s_1/x_1, \ldots, s_n/x_n] \quad \text{for} \quad e[s_1/x_1] \ldots [s_n/x_n].$$

A possibility for defining the meaning of nondeterministic data flow programs is to associate a set of functions with every data flow program P with the set $IN(P)$ of input ports and the set $OUT(P)$ of output ports. We introduce a semantic function:

$$BF: \langle net \rangle \to ([ENV(IN(P)) \to ENV(ID(P))]).$$

It is defined as follows. At first we associate with every function symbol f with arity (n, m) a set of functions by

$$FF: (\langle fid \rangle \cup \{merge\}) \to P([STREAM(ATOM)^n \to STREAM(ATOM)^m]).$$

For f in $\langle fid \rangle$ we define

$$f \in \langle fid \rangle \Rightarrow FF(f) = \{f'\},$$

$$FF[merge] = \{\lambda s1, s2 : sched(s1, s2, d) : d \in \{1, 2\}^\infty\}$$

where *sched* is defined as above. Note that

$$[STREAM(ATOM)^n \to STREAM(ATOM)^m]$$

is used to denote the set of *continuous* stream-processing functions.

With every program P:

$$[x_1 = f_1(y_1), \ldots, x_k = f_k(y_k), y_{k+1}]$$

we associate now a set of functions by

$$TF: \langle net \rangle \to P([ENV(ID(P)) \to ENV(ID(P))])$$

defined by

$$TF[P] = \{\lambda e : e[g_1(e(y_1))/x_1, \ldots, g_k(e(y_k))/x_k] : g_1 \in FF[f_1] \wedge \cdots \wedge g_k \in FF[f_k]\}.$$

Note that all f in $TF[P]$ are continuous. Now we can define

$$BF[P] = \{\lambda e0 : \textbf{fix}\ \lambda e : f(e[e0(z)/z]) : f \in TF[P]\}$$

where z is assumed to denote the tuple of input ports, and $\textbf{fix}\, f$ denotes the least fixed point of the function f. Since f is monotonic and continuous, this fixed point is well-defined and even identical to the least upper bound of the iteraction of f starting with the bottom element from *ENV*.

If we want to hide streams just used internally (treating the corresponding stream-identifiers as 'locally defined' or 'bound') we may define

$$BFH : \langle net \rangle \to P([ENV(ID(P)) \to ENV(ID(P))]),$$

specified by

$$BFH[P] = \{\lambda e\colon e[g(e)(y_{k+1})/y_{k+1}]\colon g \in BF[P]\}.$$

By this denotational semantics a nondeterminate data flow program is interpreted as repesenting a class of determinate functions for the associated graph. The correctness of this semantic definition with respect to the operational one for the associated data flow graph can be seen by the following theorem:

Theorem 6.3. *Let P be a program and g the associated graph. Let furthermore S be the set of results of computation sequences in g starting from the initial state σ_0; then*

$$\{h(\sigma_0)\colon h \in BF[P](e_{\sigma_0})\} = \{e_\sigma : \sigma \in S\},$$

where

$$e_\sigma(x) = \begin{cases} \varepsilon & \text{if } x \notin ID(P), \\ \sigma(arc(x)) & \text{if } x \in ID(P). \end{cases}$$

Proof. For every result σ of a computation sequence $\{\sigma_i\}_{i \in \mathbb{N}}$ there exists $f \in TF[P]$ such that

$$f^i(e_{\sigma_0}) = e_{\sigma_i}.$$

Since f is continuous, we have

$$(\mathbf{fix}\, f)(e_{\sigma_0}) = \mathrm{lub}\{f^i(e_{\sigma_0})\} = \mathrm{lub}\{e_{\sigma_i}\} = e_\sigma.$$

On the other hand for every $f \in TF[P]$ the states σ_i with

$$\sigma_i(\mathrm{arc}(x)) = f^i(e_{\sigma_0})(x)$$

denote a computation sequence. □

Note that in contrast to the definitions in the previous section we did not use any techniques of power domains in this section for defining the semantic function *BF*.

6.4. Comparison of nets as set-valued functions and nets as sets of functions

Now we are going to look at the basic differences between the two interpretations above. Obviously the first one allows a richer combinatorics as can be seen from the following lemma:

Lemma 6.4. *For every function symbol* $f \in \langle fid \rangle \cup \{merge\}$ *and every data flow program* P *we have*

$$(0) \quad FS[f](S) = \{g(s): g \in FF[f] \wedge s \in S\},$$

$$(1) \quad (\forall x \in ID: E(x) = \{e(x)\}) \Rightarrow$$

$$\forall x \in ID: \{h(e)(x): h \in TF[P]\} \subseteq TS[P](E)(x),$$

$$(2) \quad (\forall x \in ID: E(x) = \{e(x)\}) \Rightarrow$$

$$\forall x \in ID: \{h(e)(x): h \in BF[P]\} \subseteq BS[P](E)(x). \quad \square$$

With respect to our program *MA* showing the merge anomaly this lemma gives a simple explanation. The atom b may be a first output of the data flow program if we take the semantics defined by *BS*. But it may not be an output if we take *BF*.

The merge anomaly can also be explained in terms of parameter mechanisms for functions with nondeterminate expressions as parameters (cf. [5]). If we substitute nondeterministic expressions for identifiers, we obtain more possible results than if substituting consistently values (from the set of possible values) for identifiers. In the first case the same value is used for all occurrences of the identifier ('call-time-choice"), in the second case we can make individual choices for each of the occurrences.

In cyclic networks of communicating agents a recursively defined stream associated with an arc has to have one determinate identity in every instantiation (or computation resp.). All other arcs using the stream should use the *same* stream in one instantiation. So a nondeterministic cyclic network has to be considered as a class of deterministic networks and not as a graph where classes of streams are associated with the arcs, if we want to avoid the merge anomaly.

Basically a recursive definition with a nondeterministic functional allows (at least) two distinct interpretations:

(1) it can be seen as a fixed-point equation for a set of determinate objects or a set-valued function, or

(2) it can be seen as a set of fixed-point equations for objects or functions.

Both possibilities have been treated in this section, both give proper semantic definitions for data flow programs, but they define distinct semantics (cf. the lemma above). The first considers the recursive equations as shorthands for an infinite acyclic, sharing-free graph, the second one as shorthand for a finite, cyclic one.

Note that for both possibilities we can give consistent operational semantics also without looking at graphical representations. In the first case we can use set-valued states (similar to nondeterministic environments), in the second case the correspondence between the associated graph and the program is so close that it is obvious how rewriting rules for the programs look like (cf. [6]).

Note that we have chosen the most simple semantics that did allow us to describe and analyse the merge anomaly. For a more sophisticated semantic model which also treats the domain problem more appropriate, see [6].

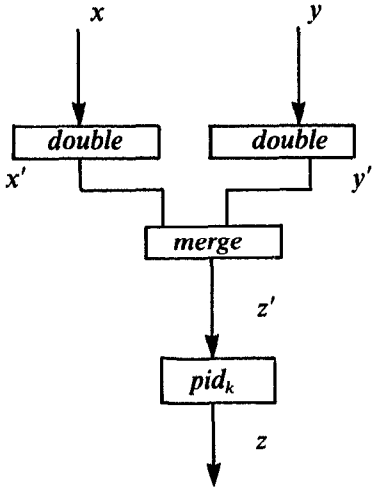
Actually as long as we consider just finite cyclic data flow graphs we do not need power domains at all, since fixed-point theory is only needed for the deterministic functions in the set of possible functions. If one wants to treat also infinite cyclic graphs (that are recursively defined) or networks with recursively defined nondeterministic functions in the nodes (as in [6]), one has to deal with the power domain over the space of continuous stream processing functions. Advanced fixed-point theory is able to cope with these problems and even with nonstrict or fair merge (see also [6]).

7. The merge anomaly of Brock and Ackermann

In [4] another example for a merge anomaly is given. We start by shortly describing their merge anomaly and then analyse how the respective data flow program is treated in our approach.

7.1. The merge anomaly

Let P_1 and P_2 be the data flow graphs as shown below:



P_1 and P_2

Syntactically, P_1 and P_2 may be written by the simple data flow program:

$$[x' = double(x),\ y' = double(y),\ z' = merge(x', y'),\ z = pid_k(z'),\ z].$$

Here *double*, pid_1, and pid_2 are determinate functions which produce at most two output values. Function *double* produces two copies of its first input value. Both pid_1 and pid_2 pass through their first two input values, but pid_1 will produce its first

output as soon as it receives its first input, while pid_2 will not produce any output until it has received two input values. The defining equations for these functions are

$$double'(\varepsilon) = \varepsilon, \qquad pid_1'(\varepsilon) = \varepsilon, \qquad pid_2'(\varepsilon) = (\varepsilon),$$

$$double'(i\&x) = i\&i\&\varepsilon, \qquad pid_1'(i\&\varepsilon) = i\&\varepsilon, \qquad pid_2'(i\&\varepsilon) = \varepsilon,$$

$$pid_1'(i\&j\&x) = i\&j\&\varepsilon, \qquad pid_2'(i\&j\&x) = i\&j\&\varepsilon.$$

Despite the difference between pid_1' and pid_2', networks P_1 and P_2 are represented by the same input/output relation, i.e. for any given pair of input streams they produce the same set of possible output streams. Neither network produces any output unless it receives input. If P_k receives one or more input values on either input port, its internal pid_k process is guaranteed to receive two or more input values, thus 'avoiding' the difference between the processes, P_1 and P_2 have the same input/output relation:

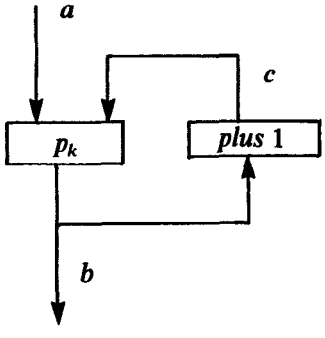
$$x = \varepsilon,\ y = \varepsilon \Rightarrow z \in \{\varepsilon\},$$

$$first(x) = i,\ y = \varepsilon \Rightarrow z \in \{i\&i\&\varepsilon\},$$

$$x = \varepsilon,\ first(y) = j \Rightarrow z \in \{j\&j\&\varepsilon\},$$

$$first(x) = i,\ first(y) = j \Rightarrow z \in \{i\&i\&\varepsilon,\ i\&j\&\varepsilon,\ j\&i\&\varepsilon,\ j\&j\&\varepsilon\}.$$

However, there is a subtle difference in their behaviors: P_2 will not produce its first output until its second output has been determined. These networks can be placed within a larger network which uncovers this difference. Let Q_1 and Q_2 be the data flow network



where *plus*1 is determined by the equation

$$plus1(x\&s) = (x+1)\&plus1(s).$$

Q_k is a cyclic network with the nondeterminate stream-processing function P_k and the determinate stream-processing function *plus*1. Inputs to Q_k are routed to the leftmost input port of P_k. The outputs of P_k are routed to two sources: to the output port of Q_k and, through the *plus*1 operator, to the rightmost input port of P_k. A possible equational specification of Q_k is

$$[x' = double(x),\ y' = double(c),\ z' = merge(x', y'),\ b = pid_k(z'),\ c = plus1(b),\ b].$$

Suppose Q_1 receives the single input value 5. The value 5 passes through the leftmost double, through *merge*, and through pid_1. The value 5 then becomes the first output of Q_1. The value 5 is also input to the *plus*1 operator, causing the value 6 to be presented to the rightmost port of Q_1, where it passes through the rightmost double. Note that *merge* has a 'choice' of producing as its second output either its second leftmost input, which is 5, or it first rightmost input which is 6. Consequently, the second output of Q_1 could be either 5 or 6. Therefore for Q_1 we obtain

$$a = 5\&\varepsilon \Rightarrow b \in \{5\&5\&\varepsilon, 5\&6\&\varepsilon\}.$$

Now suppose Q_2 receives the input sequence 5. The value 5 passes through *merge*, and enters pid_2. However pid_2 will produce no output until it has received a second input. Eventually, a second value 5 may be produced by the leftmost double and pass through the *merge* to pid_2. Then pid_2 and consequently P_2 and Q_2, will produce the output sequence $5\&5\&\varepsilon$. Therefore for Q_2 we obtain

$$a = 5\&\varepsilon \Rightarrow b \in \{5\&5\&\varepsilon\}.$$

7.2. Analysis of the anomaly

It is clear from what had been said above that the association of set-valued functions cannot work when aiming at an operational semantics as outlined above. But now let us see what happens when taking our semantic definition that associates a set of functions with every net. In $BFH[P_1]$ we have a function on environments that corresponds to the function f with

$$f(i\&e, \varepsilon) = i\&\varepsilon$$

whereas such a function cannot be found in $BFH[P_2]$. Actually it is a little bit tricky to see that f actually is in $BFH[P_1]$. One has to take $d = 1\&2\&d'$ in the definition of the *merge*.

In particular $BFH[P_1]$ and $BFH[P_2]$ are definitely distinct and correspond (according to the definitions above) to the operational semantics of the data flow graphs P_1 and P_2.

8. Implementations of data flow programs

There is a second aspect of data flow programs that is of interest in this context: Under which circumstances can we replace a data flow graph by another one (for instance obtained by unfolding or folding loops) without changing the 'correctness'.

We consider the following situation: We assume a data flow graph $P0$ that produces only (w.r.t. some specifications) correct results and is also correct w.r.t. to the composition to other data flow programs. Consequently every data flow program $P1$ is correct (and thus $P1$ can be taken for replacing $P0$) iff

$$BFH[P1] \subseteq BFH[P0].$$

This condition is in particular fulfilled if there is a graph morphism from the graph associated with PO to the graph associated with $P1$.

Lemma 8.1. *If from the data flow programs P0 and P1 there is a surjective graph morphism from the graph associated with P1 to the graph associated with P0, then*

$$BFH[P0] \subseteq BFH[P1].$$

Proof. Label in the associated graph of $P1$ every arc with the set of values that the arcs of the graph associated with $P0$ take.

If $P1$ and $P0$ are deterministic, then we have $BFH[P1] = BFH[P0]$ and so both data flow programs are equivalent and $P1$ can also be taken as implementation for $P0$. (In the case of nondeterministic data flow programs this is generally only possible, if the graph morphism is the identity for all nodes labelled with the symbol *merge*.) □

9. Comparison to other work and concluding remarks

As already mentioned the merge anomaly was first found in [8]. The proposed solution, there, however, is much to less abstract: 'partially ordered events' (cf. [8, p. 353]) are obviously very close to operational semantics.

Remarkably in the discussion at the end of Keller's presentation Vaughan Pratt said: "... the merge anomaly illustrates nothing beyond the fact that an incorrect denotational semantics may well not agree with a correct operational semantics". This is exactly our conclusion, too. What remained was to give the 'correct' denotational semantics. In [14] sets of traces in the form of 'partially ordered multisets of events' are advocated for the representation of processes. This is simply not necessary at least for coping with the merge anomaly.

Many other authors also tried to follow Keller's suggestion in one or the other sense trying to include some event structure or causality relations into the semantic domains. For instance Kosinski [9] introduces (without explicitly mentioning the merge anomaly) 'tagged streams', i.e. adds the 'decisions' of the merge function as indices to the data in streams. In [2] a similar technique is used based on 'indexed sets'. But in both cases not the merge anomaly but the general domain problem for nondeterministic stream processing functions is the motivation.

A lengthy discussion of Keller's example is found in [4]. There 'scenario-sets' are advocated, which roughly are yet another version of partially ordered event structures. A similar approach is taken in [1] where a semantic model is suggested, where "also the possible order of communications on their channels is recorded".

As it has been demonstrated in the previous sections, the merge anomaly can be avoided when using a correct denotational semantics without introducing any partially ordered event structures explicitly. The resulting semantic domain has some similarities to the techniques of [12] where oracles are used to turn nondeterminate nets into deterministic ones, leading to a set of deterministic nets.

Appendix. A denotational treatment of scenarios

Scenarios as defined in [4] establish a causality relation between input streams and output streams of a set-valued function. Let $D1$ and $D2$ be arbitrary countably algebraic domains. For a nondeterministic function

$$f: D1 \to P_{EM}(D2)$$

a scenario set for f is a family $S = \{S_y^x: y \in f(x)\}$ of mappings,

$$S_y^x \subseteq [D1_x \to D2_y] \quad \text{where } D_x = \{z \in D: z \sqsubseteq x\}$$

for $x \in D1$, $y \in f(x)$; we require for $x1 \in D1$,

$$\{g|_{D1_{x1}}: \exists x2 \in D1, y2 \in D2: x1 \sqsubseteq x2 \wedge g \in S_{y2}^{x2}\} = \{g \in S_{y1}^{x1}: y1 \in f(x1)\}.$$

Note that scenarios establish a causality relation between input and output: if $x1 \sqsubseteq x2$, then for every scenario g in the scenario set S_{y2}^{x2} by $g(x1)$ the amount of output (as an approximation of the output $y2$) is given that is caused by the (partial) input $x1$.

For $D1 = D2$ we can define fixed points on scenario sets. Let f be a nondeterministic function with scenario set S. We define the fixed point over (f, S) by the $\sqsubseteq_{EM}$-least set X fulfilling the equation

$$X = \{y \in f(y): \exists S_y^y \in S, g \in S_y^y: y = \mathbf{fix}\, g\}.$$

Note the similarity of this version of scenarios to sets of functions.

References

[1] R.J.R. Back and H. Mannila, A refinement of Kahn's semantic to handle non-determinism and communication, *Proc. ACM Conference on Principles of Distributed Computing*, Ottawa (1982) 111-120.

[2] H. Bekic, Nondeterministic programs: An example, Unpublished manuscript, 1982.

[3] F. Boussinot, Réseaux de processes avec mélange équitable: une approache du temps réel, Université Paris VII, These de Doctorat d'Etat, 1981.

[4] J.D. Brock and W.B. Ackermann, Scenarios: A model of non-determinate computation, in: J. Dias and I. Ramos, Eds., *Formalization of Programming Concepts*, Lecture Notes in Computer Science **107** (Springer, Berlin, 1981) 252-267.

[5] M. Broy, A fixed point approach to applicative multiprogramming, in: M. Broy and G. Schmidt, Eds., *Theoretical Foundations of Programming Methodology* (Reidel, Dordrecht, 1982) 565-623.

[6] M. Broy, Fixed point theory for communication and concurrency, *IFIP TC2 Working Conference on "Formal Description of Programming Concepts II"*, Garmisch-Partenkirchen, 1982.

[7] M. Broy, Applicative real time programming, in: R.E.A. Mason, Ed. *Information Processing 83* (North-Holland, Amsterdam, 1983) 259-264.

[8] R.M. Keller, Denotational models for parallel programs with indeterminate operators, in: E.J. Neuhold, Ed., *Formal Description of Programming Concepts* (North-Holland, Amsterdam, 1978) 337-366.

[9] P.R. Kosinski, Denotational semantics of determinate and nondeterminate data flow programs, MIT, Laboratory for Computer Science, TR-220, 1979.

[10] E.J. Neuhold (Ed.), *Formal Description of Programming Concepts* (North-Holland Amsterdam, 1978).
[11] M. Nivat, On the interpretation of recursive polyadic program schemes, *Symp. Math.* **15** (1975) 255-281.
[12] D. Park, The 'fairness' problem and nondeterministic computing networks, *Proc. 4th Advanced Course on Theoretical Computer Science*, Amsterdam (1982).
[13] G. Plotkin, A powerdomain construction, *SIAM J. Comput.* **5** (1976) 452-486.
[14] V.R. Pratt, On the computation of processes, *Proc. ACM Conference on Principles of Programming Languages* (1982) 213-223.
[15] M.B. Smyth, Power domains, *J. Comput. System Sci.* **16** (1978) 23-36.